A DSL for Describing the Artificial Intelligence in Real-Time Video Games

Tom Hastjarjanto

Johan Jeuring

Sean Leather

Technical Report UU-CS-2013-003

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

A DSL for Describing the Artificial Intelligence in Real-Time Video Games

Tom Hastjarjanto[1], Johan Jeuring[1,2], and Sean Leather[1]
[1]Utrecht University, [2]Open University, The Netherlands

Abstract—**Many games have computer-controlled agents that play against a player. The behavior of these computer-controlled agents is described by means of the artificial intelligence (AI) in the game. The AI is an important component of the game, and needs to be developed carefully, and adapted regularly. This paper introduces a novel language for describing the decision making process of the AI in real-time video games. We develop a declarative, domain-specific language (DSL) embedded in the functional programming language Haskell for real-time video games. We use the DSL to describe the AI of a prototype real-time video game.**

I. Introduction

The computer games industry is huge, and new titles appear on a daily basis. Developing a game involves developing many aspects, such as the physics, art, graphics, sound, and the artificial intelligence (AI). Game AI is used to produce the illusion of intelligence in the behavior of computer-controlled agents (or non-player characters, bots). In many games, the gameplay code, including the AI, is implemented in a scripting language on top of a game engine. To increase productivity in the development of games, it is important to look at how the AI is modelled and implemented. Recent research suggests that the AI should be modeled separately from aspects such as performance, memory management, and physics [24], [25], [7]. Modelling the AI separately gives fewer opportunities to introduce bugs, and better opportunities to inspect, reuse, adapt, analyze, optimize, and parallellize code, and to support concurrent behavior.

In this paper we introduce a novel domain-specific language embedded in the higher-order lazy functional programming language Haskell [23], for modelling the decision making process of the AI in real-time video games.

This paper is organised as follows. Section II reviews existing approaches to implementing AI in real-time video games. Section III introduces domain-specific languages, and describes some domain-specific languages for games. Section IV develops a novel embedded domain-specific language with which we model game AI. Section V shows how we use the DSL for game AI to select actions for computer-controlled agents. Section VI presents an example in which the language is used to model the AI of a simple video game, and Section VII concludes and describes future work.

II. Implementing AI

There are several approaches to implementing game AI. The approach to developing game AI depends on the nature of the game. For example, the state space of tic-tac-toe is much smaller than the state space in Pacman. For games with a limited number of states it is possible to explore a set of future states using the minimax algorithm, which determines the next step with the greatest payoff. The alpha-beta pruning algorithm can be used to reduce the number of states explored. Alpha-beta pruning cannot be used in games in which the state changes multiple times per second, because the number of possible states becomes too large.

Game AI determines agent *behavior*: the actions an agent takes in reaction to the game state. Game AI should support [15]:

- A variety of behaviors: an agent should react differently to different situations.
- Behavior integrity: an agent should react as a human would in the same situation.

For these requirements, a number of challenges have to be tackled:

- Coherence: can the behavior of an agent be adjusted smoothly during state transitions?
- Transparency: can a player make reasonable predictions about an agent's behavior?
- Mental bandwidth: is the behavior and implementation of an agent comprehensible by a designer?
- Usability: can the behavior of an agent be customized for different scenarios?
- Variety: can an agent do different things?
- Variability: can an agent choose between a set of applicable responses given a certain situation?

AI decisions are often modeled using tree- or graph-like data structures [21] and implemented by scripts using if-then-else statements. The following sections describe some approaches to modeling decision making.

A. Behavior Trees

The designers of Halo [15] use a behavior tree or hierarchical finite state machine (more specifically, a behavioral directed acyclic graph) to implement the AI of computer-controlled agents. In a behavior tree, a node is indexed by the state and contains a behavior. Agent behavior is determined by a path of nodes in the tree. When an agent can choose between multiple transitions from a node, the AI ranks nodes by priority and the agent moves to the node with the highest priority.

B. Goal-Oriented Action Planning

Goal-Oriented Action Planning (GOAP) is a decision-making architecture developed for the game No One Lives Forever 2 [22]. Using GOAP, we specify how decisions are taken. The key concepts of GOAP are:

- Action: a step performed by an agent. An action may have preconditions and may have an effect on the state, and takes some time to complete.
- Plan: a sequence of actions determined by an agent.
- Goal: a desired state. Based on the goal an agent can produce a plan of actions that results in reaching the goal.
- Planner: the planner takes the most relevant goal of an agent and creates a plan to go from the current state to a state satisfying the goal. The planner is similar to pathfinding, and a planner implementation may be based on the A* algorithm.

Using GOAP, state transitions are obtained as part of a plan and not hard-coded. Consequently, it is possible to find solutions to problems that were not anticipated in the initial game design. A plan to reach a goal is generated using a runtime search based on the current state and preconditions of available actions. An advantage of this approach is that actions can be coded in a smaller scope. Hard-coded plans are more difficult to maintain and less fault-tolerant than an automated, machine-generated plan constructed by the planner. GOAP also offers a solution to obtaining varied behavior. A programmer may add multiple actions to achieve the same goal, with different preconditions. The planner will then select the most suitable action based on the current state.

C. Hierarchical Task Networks

A hierarchical task network (HTN) [16] offers an alternative approach to planning and executing strategies. An HTN specifies various compound high-level tasks, which each consist of a set of subtasks. The leaves of the hierarchy are primitive tasks. An HTN contains methods that describe how to reach the current goal, and the prerequisites and subtasks necessary to complete the goal. An HTN has advantages from both a planning and scripting perspective. The plans can be calculated automatically using a planner component based on the prerequisites of subtasks to achieve a goal. The scripts containing the plans are generated offline by using the results from the planner. The scripts can be put in the game engine to run at runtime using the current state of the game as input.

This is not a complete overview of approaches to modelling AI in games. Other recent work includes work on statecharts [3], [4], and embedded scripting languages [2].

For each of the approaches described above, the decision-making model is translated to software. The software obtained is integrated with the other software components of the game. Thus, every change in decision making requires adapting the model, translating the model to software, and integrating the software with the other components.

III. Domain-Specific Languages for games

A domain-specific language (DSL) is a type of programming language (or specification language in software development and domain engineering) dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. A DSL can allow a problem or solution to be expressed more clearly than a general-purpose language, especially if that type of problem reappears sufficiently often. DSLs introduce special notation for domain-specific concepts [20].

To avoid developing a complete tool chain for a new DSL, designers often embed a DSL in a sufficiently powerful host programming language that can model the DSL abstractions [14]. The existing tool chain for the host programming language can then be used to analyse and type-check software written using the DSL, and the host language can be used to implement aspects that cannot be modelled in the DSL. Code generation and documentation capabilities come for free from the host language.

Furtado, et al. [6] describe an approach to developing DSLs for games. First, decide on the target game subdomain. Then, select several example games which are used to identify the common features of the subdomain. Implement the DSL for these features, and validate the result by implementing the example games in the DSL.

In general, using a DSL makes it easier to inspect, reuse, adapt, analyze, optimize, and transform programs. Game-play-specific code is carefully tuned and often rewritten in the development of games, which is more easily done if the game AI is developed in a high-level DSL. With a DSL for game AI, a developer can concentrate on developing the AI and does not need to be concerned with performance, compilation, or other orthogonal aspects when developing the AI.

A number of domain-specific languages are used to describe the AI in games. Many game engines contain a scripting component tailored to producing game-play-specific code. Often, these languages are general-purpose languages, such as Lua (World of Warcraft) or Javascript (the Unity game engine), which interact with the low-level game engine. The low-level game engine, which contains functionality for networking, rendering, sound, input, math and physics, is often developed in C++. Sometimes, the scripting component offers a custom-designed scripting language, such as Unrealscript for the Unreal Engine[1], and Game-Oriented Object Lisp (GOOL) [7].

IV. A DSL for Game AI

To make it worthwhile to develop a DSL for game AI, the DSL should satisfy a number of requirements:

- The DSL is general: the language can be used to describe (sequences of) actions in any video game, with any kind of state. This ensures that the DSL can be used for many games.

[1]http://udn.epicgames.com/Three/UnrealScriptReference.html

- We can interpret game AI written in the DSL to calculate the next action of an agent based on the AI description and the state.
- Game AI expressed in the DSL is readable by game AI developers. If the game AI is transformed, this happens behind the scenes, and can only be adapted by adapting the transformation process, not by updating the transformed AI.
- The DSL is compositional: game AI can easily be combined to form new game AI or reused as part of other game AI.

These requirements are partially derived from requirements for specifying how to solve interactive exercises in learning environments [11]. The kind of interactivity of learning environments is not unlike that of games. We have not found DSLs for describing game AI that clearly satisfy the last two requirements. This paper develops a DSL that satisfies all of the above requirements.

We want to develop an embedded DSL for game AI with which we can implement the behavior of computer-controlled agents in real-time video games such as Pacman, Space Invaders, etc. These games have a number of characteristics:

- A game may have multiple computer-controlled agents.
- An agent performs actions.
- Actions of different agents in a game may happen concurrently and influence a single state.
- Time plays a crucial role in many actions: a bullet (usually) travels faster than a running character.
- The game state may be updated multiple times per second.
- Actions may be interrupted: for example, the actions of other agents may have made them unnecessary or impossible.

The lazy, higher-order, functional programming language Haskell [23] offers excellent possibilities to define embedded DSLs using higher-order domain-specific combinators [14]. Furthermore, there exist several libraries for Haskell for binding with external components useful for games, such as Hipmunk and GLUT. For this reason we decided to embed our DSL in Haskell.

The AI of an agent is specified by a value of the Haskell datatype *GameAI*, which defines the core language with a set of constructors:

```
data GameAI s
   = Action (s → s)
   | Idle
   | GameAI s  :⋆:  GameAI s
   | GameAI s  :|:  GameAI s
   | GameAI s :|>: GameAI s
   | Fix (GameAI s → GameAI s)
```

The datatype *GameAI* abstracts over the type s of the game state, which may take very different forms for different games. The basic element of game AI is an *Action*. An *Action* takes as an argument a function of type $s \rightarrow s$, which transforms the state in some way. For example, the actions of shooting a gun or detecting an enemy can be modelled by:

```
data GameState = ...
shoot :: GameAI GameState
shoot = Action ...
detectNearestEnemy :: GameAI GameState
detectNearestEnemy = Action ...
```

The exact implementation of an action on the state is part of the game-specific implementation of a game and is omitted.

The *Idle* constructor represents AI that does not perform any action. This constructor may not look very useful, but it serves a purpose when combining *GameAI*.

We combine *GameAI*s to construct more advanced *GameAI*s using combinators. The binary *sequence* combinator (:⋆:) is used to specify that a (sequence of) action(s) a is followed by another (sequence of) action(s) b: a :⋆: b. *Idle* is an algebraic zero of :⋆:, and *Succeed* a unit. The binary *choice* combinator (:|:) is used to specify that an agent can choose between a (sequence of) action(s) a and another (sequence of) action(s) b: a :|: b. The interpretation of a :|: b randomly chooses between a and b. If we want to try the left argument a before the right argument b, we use the left-biased choice a :|>: b instead. *Idle* is a unit of :|: and :|>: . The following code shows how to implement combined game AI for moving in two different ways to a flag using atomic move actions.

```
move :: (Float, Float) → GameAI GameState
move (x, y)  = Action ...
moveToFlag :: GameAI GameState
moveToFlag =    move (20, 30)
             :⋆: move (50, 60)
             :⋆: move (70, 80)
            :|:  move (20, 60)
             :⋆: move (70, 80)
```

Note that :⋆: binds stronger than :|: .

The *Fix* combinator introduces recursion in the game AI language. It can be used to repeat an action multiple times (*many*), or until a certain condition is satisfied.

```
many          :: GameAI s → GameAI s
many s        = Fix (λx → Action id :|: s :⋆: x)
shootEnemies :: GameAI
shootEnemies = many  (  detectNearestEnemy
                        :⋆: shoot
                        )
```

In addition to the basic constructs defined above, we include support for *interleaving* actions [11]. Interleaving is necessary when describing several separate (sequences of) actions(s) for which the order does not matter. We extend the core language with the following constructors:

```
data GameAI s
   = ...
   | Atomic (GameAI s)
```

$$\begin{array}{l}
\mid\ \mathit{GameAI}\ s\ :\!\|\!:\ \mathit{GameAI}\ s\\
\mid\ \mathit{GameAI}\ s\ :\!\underline{\|}\!:\ \mathit{GameAI}\ s
\end{array}$$

The *Atomic* constructor prevents a game AI from being interleaved with other game AI for the same agent. The interleave construction $a\ :\!\|\!:\ b$ allows the (non-atomic) actions from a and b to be interleaved in any order. For example, if we want to capture a flag and shoot enemies, but the order in which we perform these actions does not matter, we declare

$$\begin{array}{lll}
\mathit{goal} & :: & \mathit{GameAI}\ \mathit{GameState}\\
\mathit{goal} & = & \mathit{shootEnemies}\\
 & :\!\|\!: & \mathit{moveToFlag}\ :\!\star\!:\ \ldots
\end{array}$$

The left-interleave construction $a\ :\!\underline{\|}\!:\ b$ first performs an action from a and then interleaves actions from the rest of a and b.

A game can have multiple agents performing actions concurrently, with a different *GameAI* value for each agent. To allow for asynchronous responses to actions (which can take variable lengths of time), we use an event-based system. An event is fired whenever one of the agents in the game may need to reevaluate its actions. Examples of events are "goal completed," "enemy detected," and "agent shot." Our datatype *GameAI* requires each agent to "know" the result of a game state update: the construction *Action* f, where f has the type $s \rightarrow s$, only allows synchronous (pure) updates to the state, since the result of the update must be immediately available to the acting agent.

Haskell functions of the type $\mathit{GameState} \rightarrow \mathit{GameState}$ are pure and cannot involve randomness or network communication or call C++ functions that change the state. The asynchronous event system and the typically imperative game components (such as the physics engine) dictate that the game state cannot be updated with a pure function. However, Haskell does have a loophole for all of these things called the *IO* monad. A function of the type $\mathit{GameState} \rightarrow \mathit{IO}\ \mathit{GameState}$ allows for an update to the state that may involve side effects.

With an event-based system, it is important to be able to analyze actions. But we cannot inspect an action if it is defined as a function; the only thing we can do with a function is apply it to an argument. So, we relax the definition of an action to allow for a non-function type. For example, the following datatype supports the actions we have already described:

$$\begin{array}{l}
\mathbf{data}\ \mathit{Action}\\
\quad = \mathit{Move}\ (\mathit{Float}, \mathit{Float})\\
\quad\ \mid\ \mathit{Shoot}\ (\mathit{Float}, \mathit{Float})\\
\quad\ \mid\ \mathit{DetectNearestEnemy}\\
\quad\ \mid\ \ldots
\end{array}$$

This has the added advantage that we separate actions from their implementation. Now that an action is a value of a datatype instead of a function, we need to specify how an action is performed. For this purpose we introduce a class *PerformAction*, with a single method *perform* of type

$$\mathit{perform} :: a \rightarrow s \rightarrow \mathit{IO}\ ()$$

To use a datatype *Action* in a game, we need to provide an implementation of *perform* for it in an instance of the class *PerformAction*.

To abstract over both the state (s) and the action (a) types, we rename the datatype to $\mathit{GameAI}\ s\ a$.

Asynchronous events also mean that an agent may need to "change its mind." For example, consider what needs to happen when agent A shoots agent B while B is running towards a goal. If B is shot but not killed, it may decide to stop running and return fire. An action must be interruptible, and an agent may run its strategy again to obtain a new sequence of actions. Consequently, the type $\mathit{Action} \rightarrow \mathit{IO}\ (\mathit{GameAI}\ s\ a)$ is more appropriate for an agent to act on.

With the above insights in mind, we replace the constructor *Action* in the *GameAI* datatype by a constructor *Act* for describing impure actions, and a constructor *Succeed* for describing pure actions:

$$\begin{array}{l}
\mathbf{data}\ \mathit{GameAI}\ s\ a\\
\quad = \mathit{Act}\ (s \rightarrow \mathit{ActionContext} \rightarrow \mathit{IO}\ (\mathit{GameAI}\ s\ a))\\
\quad\ \mid\ \mathit{Succeed}\ a\\
\quad\ \mid\ \ldots
\end{array}$$

One useful addition is *ActionContext*, a finite map from strings to arbitrary values, which allows the function to query any extra, non-state data. For example, we can use the *ActionContext* to remember which agent we tried to shoot in a previous action. Since the type of this data may vary widely, we use the type *Dynamic* for the values:

$$\mathbf{type}\ \mathit{ActionContext} = \mathit{Map}\ \mathit{String}\ \mathit{Dynamic}$$

Succeed x now includes an action x.

When an agent is under attack, it might be better to eliminate the enemy instead of continuing to move towards a flag. To constrain actions, we extend our DSL with a conditional combinator:

$$\begin{array}{l}
\mathbf{data}\ \mathit{GameAI}\ s\\
\quad = \ldots\\
\quad\ \mid\ (s \rightarrow \mathit{ActionContext} \rightarrow \mathit{IO}\ \mathit{Bool})\ :\!?\!:\\
\qquad\quad \mathit{GameAI}\ s\ a
\end{array}$$

The above example can be written as the following *GameAI*:

$$\begin{array}{rlll}
 & \mathit{underAttack} & :\!?\!: & \mathit{shootEnemies}\\
:\!|\!>\!: & \mathit{flagAvailable} & :\!?\!: & \mathit{moveToFlag}\\
:\!|\!>\!: & & \ldots &
\end{array}$$

If the *underAttack* condition evaluates to *False*, the next condition (*flagAvailable*) is evaluated, and if it holds the *moveToFlag* actions are performed. The condition uses the *IO* monad since it needs to look at the current state and possibly use information from the physics engine to evaluate its condition.

Different actions may take a different amount of time to complete. In the central game loop, we schedule the actions of the different agents based on their time. If an action of one agent completes while another agent is still performing its action, the first agent will start on its next action.

Since the state is updated multiple times per second, at the lowest level it does not matter how actions are ordered, since different orders cannot be perceived by a player.

The *GameAI* DSL satisfies the requirements listed at the beginning of this section. Since both the game state and the type of actions are type parameters of *GameAI*, the *GameAI* is general in the sense that it can be used to describe (sequences of) actions in any video game with any kind of state. As we will show in the following section, we can calculate the next step of an agent based on a *GameAI* description and the state. *GameAI* is easily readable since the AI is specified by means of constructors from a datatype which can be shown, inspected and pattern matched. As a consequence, it is easy to transform *GameAI*, for example to optimize or parallelize it. Finally, the examples given in this section show that *GameAI* is compositional: we can reuse previously define game AI (such as *moveToFlag*) in other game AI (such as the extended game AI for an agent under attack above).

V. Implementation

The DSL in the previous section describes the game AI for a computer-controlled agent. To use the *GameAI* we define functionality to select actions given a *GameAI* description and the current state.

Function *firsts* is the central function for dealing with *GameAI*. It returns a list of actions that can be taken as the first step by the computer-controlled agent. We define a simplified version of *firsts* to show the central idea behind it. The function *firsts* takes a *GameAI* description as argument, and returns a list of pairs consisting of a possible action, together with the remaining *GameAI* descriptions:

```
firsts :: GameAI s a → [(a, GameAI s a)]
firsts g = [ (r, x :⋆: y)
           | Atomic (r :⋆: x) :⋆: y ← split g
           ]
```

The function *firsts* uses a function *split*, which takes a *GameAI*, and returns all ways it can be split into an initial atomic part and a remaining part. An atomic sequence of actions cannot be interleaved with other actions. By definition, each single basic action is performed atomically. We then take the first action of the atomic part. The definition of *split* is omitted, but is very similar to the definition of the same function for rewrite strategies for interactive exercises [11]. We also have a variant of *firsts* that takes context information into account.

Sometimes, state changes are relevant for more than one agent. For example, when an agent captures a flag, the game sends an event to all agents to notify that the flag has been captured. For some agents, this might mean that they have to reconsider their actions. The code for performing an action fires an event as soon as an action completes. The datatype *Event* distinguishes between a *CompletedEvent* and an *InterruptEvent*. An event specifies the recipients of the event with a list *EventAgents*.

```
data Event       = CompletedEvent EventAgents
                 | InterruptEvent EventAgents
data EventAgents = EventAgent [String]
                 | All
```

To keep track of events, we use an *EventChannel*, a list of unhandled events in a game. In its simplest form, the function *fireEvent* inserts an event in the channel.

```
type EventChannel = [Event]
fireEvent :: EventChannel → Event → EventChannel
fireEvent evChan event = event : evChan
```

However, the game loop uses the event channel, updating it continuously to handle asynchronous events. Thus, we maintain a reference to the event channel in the *IO* monad using an *IORef*. We use the following modified version of *fireEvent*:

```
fireEvent' :: IORef EventChannel → Event → IO ()
fireEvent' evChanRef event = do
  evChan ← readIORef evChanRef
  writeIORef evChanRef (fireEvent evChan event)
```

The function *fireEvent′* is used in the game-play code to fire events when agents complete their action or when external events happen that influence the game AI of agents.

We have a component that listens to incoming events and responds to state changes by notifying agents. The function *processEvents* is called by the game loop, which runs every frame. The function calls the appropriate response handler to respond to changes in the state that influence the actions of the agents. The response handler is implemented as the *eventResponse* function which is called with the function *next* or *interrupt* given below. In the following definitions, the implementations of *eventResponse* and *State* both depend on the game.

```
processEvents :: EventChannel → State → IO State
processEvents []       state = return state
processEvents (x : xs) state = do
  state' ← processEvent x state
  processEvents xs state'
processEvent :: Event → State → IO State
processEvent (CompletedEvent e) state =
  eventResponse e completed state
processEvent (InterruptEvent e)  state =
  eventResponse e interrupt state
```

The function *eventResponse* takes as arguments a list of agents involved, a function (*completed* or *interrupt*), and a state, and it calculates the next actions of the agents involved. In the case of an interrupt event, the current action from the *GameAI* of an agent is cancelled, and a new first action is calculated. In the case of a completed event, we calculate the next action of the agents. The implementation of the functions *eventResponse*, *completed*, and *interrupt* is omitted here and can be found in the first author's MSc thesis [9].

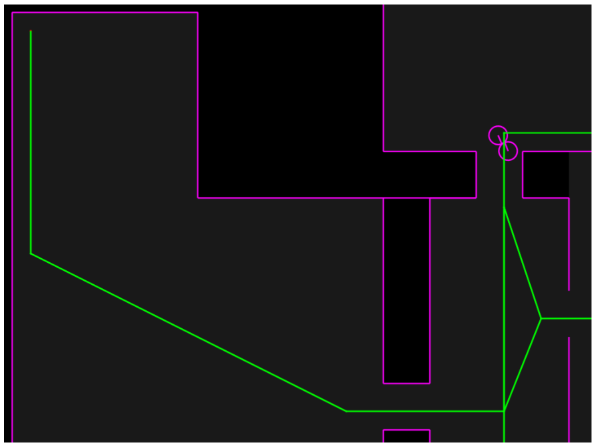

Fig. 1. Screenshot of the (counter-)terrorists game

VI. Example Game

We have implemented a small 2D real-time action game that contains agents that use $GameAI$ to compute actions to complete the game. The game is played by a team of *terrorists* who want to capture a flag and a team of *counter-terrorists* who want to prevent this. The game area contains a spawning location for terrorists, a spawning location for counter-terrorists, and two locations where a flag can be captured. The screenshot below shows part of a game window. Players are rendered as circles, walls as purple lines and the navigation structure as green lines.

The game has been implemented[2] in Haskell using the OpenGL library to render the game on screen and to schedule the game loop. On top of OpenGL, we use GLUT which provides input handling. The physics calculations are performed using the Chipmunk physics library via the Haskell binding Hipmunk. The game $State$ contains information about the Chipmunk space, and the various physical objects.

A. Actions

The agents in our game can perform a number of basic actions, such as shoot, move, and take the flag. We have an $Action$ datatype in which the different basic actions are listed.

B. Game AI

The terrorists want to capture the flag, and the counter-terrorists want to prevent that from happening. The main game AI for all computer-controlled agents takes the position of the flag as argument and returns the $GameAI$.

```
mainGameAI     :: Pos → GameAI State Action
mainGameAI pos = combat :|>: moveToFlag pos
```

To capture or defend the flag, agents try to shoot the members of the other team. This is described by the game AI $combat$.

```
combat = enemiesInTheSameRoom :?:
    (  shootNearestEnemy
    :|: throwGrenade
    :|: takeCover
    )
```

The function $enemiesInTheSameRoom$ checks the state to determine if there are any enemies in the same room. If this condition holds, the game AI following :?: is performed. This game AI randomly chooses between the actions to shoot the nearest enemy, throw a grenade, or take cover. The definitions of $enemiesInTheSameRoom$, $shootNearestEnemy$, $throwGrenade$ and $takeCover$ are omitted. The final action generated by these functions fires a $CompletedEvent$ when it has been completed. If there are no enemies in the room, $combat$ fails, and the $moveToFlag$ game AI is used to move to the location of the flag.

Both teams start in locations from which they need to move to the location of the flag. An agent moves to a location by navigating via several waypoints in the map to reach a target. We can either manually construct this path, or use a path finding algorithm in our game AI, as is usually done in video games. We have implemented the A* path finding algorithm in the function $shortestPath$ to generate a list of waypoints to navigate to a target destination. This also demonstrates that we can easily embed other technologies in our game AI. The function $moveToFlag$ takes a position and uses function $moveToWaypoint$ to construct game AI that consist of a sequence of move actions to the waypoints on the shortest path to the flag.

```
moveToFlag :: Pos → GameAI State Action
moveToFlag pos =
  let moves waypoints =
        case waypoints of
          [p]      → arriveAtTarget p
          (p : ps) → moveToWaypoint p :⋆: moves ps
  in Act
       (λstate context →
          moves (shortestPath state pos)
       )
```

The function $moveToFlag$ is a slightly simplified version of the function used in the implementation of the game, but it shows the essential aspects. The definitions of $moveToWaypoint$ and $arriveAtTarget$ are omitted. $arriveAtTarget$ is an action that fires a $CompletedEvent$.

The function $mainGameAI$ is not a complete description of the game AI. We also have to implement game AI for returning to base when the flag is captured by a terrorist, and for counter-terrorists to try to retrieve the flag when the flag is captured by a terrorist, or defend the flag. Both teams need to react in case the flag is captured. This is triggered by an $InterruptEvent$ for all agents. Because the game AI is similar for both teams except for the game AI after the flag has been captured by a terrorist, we can take advantage of the combinators and the fact that we can directly reuse values of $GameAI$ to simply create new game AI.

[2]Source download: http://intellicode.nl/thesis.html

VII. Conclusion and Future Work

A. Conclusion

This paper describes a domain-specific language for describing artificial intelligence in real-time video games. The DSL offers domain-specific notation to specify game AI. We have chosen to embed the DSL in Haskell, giving us an implementation for free, and access to the full functionality of a programming language. The design of our DSL builds upon previous work on a DSL for describing interactive exercises [10], [12], [11], and offers constructs similar to those present in behavior trees and hierarchical task networks. The DSL for describing interactive exercises has been adapted in various ways.

- An event system has been added to facilitate the communication between various agents and to implement the effect of actions from one agent on other agents.
- The DSL has a number of constructors useful for describing game AI. In particular, we use the combinator $:?:$ to describe conditional actions, and the Act constructor to describe actions that need information from the state, and possibly an action-specific context.
- The basic components produced by game AI are *action*s, which have a duration attached.
- Using the game AI, we *generate* actions of computer-controlled agents, instead of parsing steps.
- The game-play code applies actions to the state. This implies that the handling of actions can take advantage of other components such as physics or animation libraries which are accessible from the game play code.

The example game in Section VI shows how we successfully used this DSL to describe and implement the game AI of computer-controlled agents in a real-time video game.

Our DSL can be an attractive alternative to existing methods for specifying game AI that mostly rely on using general-purpose control flow elements or use tree- or graph-like data structures. Our DSL offers a convenient and concise way to describe, include, transform, and reuse strategies in real-time video games, and does not require a graphical user interface to conveniently adjust, prototype, and maintain the game AI. Furthermore, our game AI DSL can be directly executed.

B. Future Work

We want to investigate if we can use our DSL to describe other components of a game too, such as animations, the story line, or the behavior of groups of agents.

We want to further experiment with our DSL by implementing other kinds of games. In the near future, we hope to develop a serious game for practicing communication skills.

To illustrate the power of our DSL, we want to implement behavior trees, GOAP, and HTN, in our DSL.

At the moment the game AI and action implementors are responsible for firing an event upon completion of a series of actions. The events list the agents that should be notified. Future work will focus on the relationship between actions and events, to determine if we can further integrate events and actions to remove the need for firing events and listing agents involved.

References

[1] E.M. Avedon. *The structural elements of games*, In The Study of Games, John Wiley, 1973.

[2] A. Calleja and G. Pace. *Scripting Game AI: An Alternative Approach using Embedded Languages*, Proceedings WICT 2010, University of Malta, 2012.

[3] C. Dragert, J. Kienzle, and C. Verbrugge. *Toward High-Level Reuse of Statechart-based AI in Computer Games*, Proceedings GAS'11, pages 25-28, ACM, 2011.

[4] Christopher Dragert, Jorg Kienzle, ans Clark Verbrugge. *Statechart-Based AI in Practice*, Proceedings AIIDE'12, AAAI Press, 2012.

[5] T. Fullerton. *Game design workshop: a playcentric approach to creating innovative games*, Morgan Kauffmann, 2008.

[6] A.W.B. Furtado, A.L.M. Santos and G.L. Ramalho. *SharpLudus revisited: from ad hoc and monolithic digital game DSLs to effectively customized DSM approaches*, Proceedings of the SPLASH '11 Workshops, pages 57-62, ACM, 2011.

[7] A. Gavin. *Making the solution fit the problem: AI and character control in Crash Bandicoot*, Computer Game Developers Conference Proceedings, 1997.

[8] A. Gerdes, B. Heeren and J. Jeuring. *Properties of exercise strategies*, Proceedings of IWS 2010: 1st International Workshop on Strategies in Rewriting, Proving, and Programming. Electronic Proceedings in Theoretical Computer Science 44, pages 21-34, 2010.

[9] T. Hastjarjanto *Strategies for real-time video games*, MSc. thesis, Computing Science, Utrecht University, to appear, 2013.

[10] B. Heeren and J. Jeuring. *Recognizing strategies*, In A. Middeldorp (Ed.), Proceedings of WRS 2008: 8th International Workshop on Reduction Strategies in Rewriting and Programming, Electronic Notes in Theoretical Computer Science, Volume 237, pages 91-106, 2009.

[11] B. Heeren and J. Jeuring. *Interleaving strategies*, In J.H. Davenport et al (Eds.), Proceedings of Calculemus/MKM 2011, LNAI 6824, pages 196-211, Springer, 2011.

[12] B. Heeren, J. Jeuring and A. Gerdes. *Specifying Rewrite Strategies for Interactive Exercises*, Mathematics in Computer Science, 3 (3), pages 349-370, 2010.

[13] H. Hoang, S. Lee-Urban and H. Muñoz-Avila. *Hierarchical plan representations for encoding strategic game AI*, Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference, AAAI Press, 2005.

[14] P. Hudak. *Building domain-specific embedded languages*, ACM Computing Surveys - Special issue: position statements on strategic directions in computing research, 28 (4), 1996.

[15] D. Isla. *Handling Complexity in the Halo 2 AI*, Proceedings of the Game Developers Conference, 2005.

[16] J.P. Kelly, A. Botea and S. Koenig. *Offline planning with hierarchical task networks in video games*, Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference, 2008.

[17] R. Laemmel, E. Visser and J. Visser. *The essence of strategic programming*, draft, Computing Science, VU Amsterdam, 2002.

[18] D. Leijen. *The λ Abroad*, Ph.D. Thesis, Computing Science, Utrecht University, 2003.

[19] D. Leijen, E. Meijer and J. Hook. *Haskell as an automation controller*, In 3rd International Summerschool on Advanced Functional Programming, LNCS 1608, pages 268-288, Springer, 1999.

[20] M. Mernik, J. Heering and A. M. Sloane. *When and how to develop domain-specific languages*, ACM Computing Surveys, 37 (4), pages 316-344, 2005.

[21] A. Nareyek. *AI in computer games*, ACM Queue, 1 (10), pages 58-65, 2004.

[22] J. Orkin. *Applying Goal-Oriented Action Planning to Games* In Steven Rabin (Ed.), AI Programming Wisdom 2, pages 217-229, 2003.

[23] S. Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report*, Cambridge University Press, Cambridge, England, 2003.

[24] T. Sweeney. *The next mainstream programming language: A game developers perspective*, Invited talk at POPL '06: the 33rd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages Languages, 2006.

[25] W. White, C. Koch, J.Gehrke and A. Demers. *Better scripts, better games*, Communications of the ACM, 52 (3), pages 42-47, 2009